Variable Binding Semantics

Variable binding semantics describe how the name of a variable is related to the memory that stores its value

languages often mix-and-match their approaches! For example, C++:

- uses **value semantics** for primitive data types
- uses **reference semantics** with the reference operator (ex: `int &r = x;`)
- uses **object reference semantics** when explicitly referencing/dereferencing pointers

Value Semantics

Each variable name is directly bound to the storage that represents that variable.

- In languages like C++ (which uses value semantics for primitive data types), local variables are stored on the stack in an "activation record"
- **activation record** a map from variable names to their direct values; entire record lives in memory

```
// activation record: nothing!
int main() {
  int x = 5; // activation record: (x,5)
  int y = 0; // activation record: (x,5), (y,0)
  y = x;     // activation record: (x,5), (y,5)
}
// activation record: nothing!
```

- When reassignment is done, **the value is completely copied over**
- After `y = x`, there is no relationship between `y` and `x` (other than their values being equal)
- Not true for other binding semantics!
- for non-primitives, you'd have to copy *every* field (deep copy), expensive!

Reference Semantics

- **Reference semantics** allows you to assign an *alias name* to an existing variable and read/write it thru that alias
- The reference variable functions identically to the previous variable
- really only relevant when you have *multiple* variables.

```
int main() {
  int x = 5;
  int &r = x;  // r is a reference
  r += 1;
  // *both* x and r have the value 6! and have same address
}
```

`&r` isn't a pointer to `x`; it refers to the **same memory location** as `x`! When we update `x`, we also update `r` (and vice-versa).

Under the hood, references are often implemented with pointers - this is how it works in C++!

Object Reference Semantics

Object reference semantics binds a variable name a pointer that itself points to an object or value

Python Example

- small integer literals like `5` and `20` are objects **allocated on the heap**; they aren't created and recreated every time a new variable is made.
- the activation record maps variables to pointers to items on the heap

```
def main():
  x = 5     # x points to a 5 on the heap
  y = 20    # y points to a 20 on the heap
  y = x     # y is re-pointed to 5 on the heap; no values are copied!
            # instead, the *pointer* value for x is copied to y
            # the 20 can now be garbage collected
  y = y + 1 # y now points to a 6 on the heap;
            # x is unchanged!
```

- on `y = x`, the **pointer values are copied**, but **not the memory itself**.
- on `y = y + 1`, `x` is unaffected, since we create a new object and change `y`'s pointer; not the underlying object!

Take a look at a more complicated object with fields:

```
class Nerd:
 def __init__(self,name,iq):
  self.name = name
  self.iq = iq

 def study(self):
  self.iq = self.iq + 50

 def iq(self):
  return self.iq

 def myname(self):
  return self.name

def main():
 n1 = Nerd("Carey", 100) # points to a Carey nerd on the heap
 n2 = Nerd("Paul", 200)  # points to a Paul nerd on the heap
 n2 = n1                 # changes n2 to point to the Carey nerd;
                         # the Paul nerd can be GC'd!
 n1.study()              # changes the Carey nerd's IQ to 150
 print(f"{n2.myname()}'s IQ is {n2.iq()}") # n2 points to the Carey Nerd, so "Carey's IQ is 150"!
```

The confusion here is:

- `n2 = n1` doesn't change *any* underlying objects; it's just a pointer copy
- however, `n1.study()` changes a **field inside an object**.
  - a `150` is put on the heap, and `self.iq` now points to it

- so, we've changed the field of an object through object reference!

The difference in how object reference semantics treats object assignment (`n2 = n1`) and field access (`n1.study()`) is important!

With this in mind: many languages use object reference semantics. Some use it for everything, like Python - others, like Java, only use it for complex objects (and value semantics for primitives).

Equality

- **Object Identity**: Do two object references refer to the same object at the same address in RAM?
- **Object Equality**: Do two object references refer to objects that have equivalent values (even if they're different objects in RAM)?

C++ Pointers

- when using `*` and `&` to interact with pointers, we change the underlying values
- without `*` and `&`, we're changing the object references!

```
int main() {
  int x = 5, y = 6;
  int *px = &x; // pointer to the same memory block as x
  int *py = &y; // pointer to the same memory block as y

  *py = *px;    // changes the underlying memory, but not the pointers!
  py = px;      // changes the pointers, but not the underlying memory!
  *py = 42;     // changes the underlying memory, but not the pointers!
}
```

Name and Need Semantics

Name semantics binds a variable name to a pointer that points to an expression graph (implemented with lambda functions) called a "thunk".

Consider the Haskell expression:

```
square x = x*x
c = square (a + b)
```

We can draw the following graph to illustrate how the values are related to each other:

```
graph TD;
    c-->square;
    square-->+;
    +-->a;
    +-->b;
```

When a variable's value is needed (e.g., to be printed), the expression represented by the graph is "lazily evaluated" and a value is produced.

"Need Semantics" memoize (cache) the result of each evaluation to eliminate redundant computations.

Here's one example where this memoization can save us computation time:

```
square x = x*x
c = square (1 + 2)
d = square (1 + 2) + 3
e = square (1 + 2) + 3 + 4
```

We can note that there are some redundant computations:

- `1 + 2`
- `square (1 + 2)`
- `square (1 + 2) + 3`

When we create an expression graph, there will be **redundant subgraphs**. Here are the compact versions of them (think about why they're equivalent!):

```
graph TD;
    c-->square;
    square-->+;
    +-->1;
    +-->2;
```

```
graph TD;
    d-->+;
    +-->c
    +-->3
```

```
graph TD;
    e-->+;
    +-->d
    +-->4
```

Note: memoization is easy if expressions return the same thing every time you call them - which is only true if your expressions *have no side effects* and *immutable values*, like in Haskell!

Parameter Passing Semantics

- Pass by object reference: The formal parameter is a pointer to the argument's value/object
- Pass by name: The parameter points to an expression graph that represents the argument

Pass by Value

- Each argument is first evaluated to get a value, and a *copy* of the value is passed to the function for local use.

Pass by Reference

- *Secretly pass the address* of each argument to the function
- In the called function, all reads/modifications/assignments to the parameter are directed to the data at the original address; treat as an alias

```
void foo(Obj &n) {
  n = new Obj("B");
  n = t;
}

int main() {
  Obj o = new Obj("A");
  foo(o);
  cout << o.to_string();   // prints B
}
```

Aliasing

```
void filter(set<int> &in, set<int> &out) {
  out.clear();
  for (auto x: in)
    if (is_prime(x)) out.insert(x);  // in and out refer to same
}

int main() {
  set<int> a;
  ... // fill up a with #s
  filter(a, a);
}
```

- **aliasing** - when two parameters (maybe unknowingly) refer to the same value/object
  - may unintentionally modify it, causing bugs

Pass by Object Reference

- All values/objects are passed by (copy of the) pointer
- The called function can use the pointer to read/mutate the pointed-to argument.

```
def foo(r):
  r.set_val("Z")
  r = Obj("B")   # doesn't modify o in main

def main():
  o = Obj("A")
  foo(o)
  print(o.val)   # Z
```

- `foo(r)` copies the object reference `o` into the formal parameter `r`
- In `foo`, the `r` points to our original object, which can be mutated through the object reference.
- But we can't use assignment to change the value of the original value/object (unlike pass by reference)
  - only the *local* object reference gets assigned, the object defined in main is still there and referred to by `o`
  - assignment of object references never change the passed-in value/object, just change where the local object reference points to

Pass by Name/Need

Each parameter is bound to a pointer that points to an expression graph (a **thunk**) which can be used to compute the passed-in argument's value.

Here, a trunk is typically implemented as a lambda function, which can be called to evaluate the full expression graph and produce a concrete result when it is needed.

Haskell passes by need

```
func2 y =
  y^2+7

func1 x =
  func2 (3 + x)

main = do
  let z = func1 5
  print z
```

In pass-by-need, once an expression graph is evaluated, the computed result is *memoized* (cached) to prevent repeat computation. In the code above, if we do another `print z` after the existing print, the value of `z` will be cached without the need for another evaluation.

Practice: Classify That Language: Parameter Passing

```
def addIfFirstEven(a: => Int, b: => Int): Int =
  var sum = a
  if (a % 2 == 0)
    sum += b
  return sum

def triple(x: Int): Int =
   var trip: Int = x*3
   println("triple")
   return trip

object Main {
 def main(args: Array[String]): Unit =
   var v1 = 1
   var v2 = 2
   var result = addIfFirstEven(v1,triple(v2))
   println("The result is: "+result) //prints "The result is: 1"
}
```

- pass-by-name since the second argument of addIfFirstEven (`triple(v2)`) is only evaluated if the first argument (`v1`) is even. Since it's odd, the `triple` function never runs!
- if `v1` was even, "triple" would be printed
  - if the final print was duplicated, "triple" would be printed twice because `result` is evaluated twice

    - if using need-semantics, "triple" would only be printed once because `result` is memoized

Parameters

Actual and Formal Parameters

- **formal parameters** - arguments in the definition of a function
- **actual parameters** - arguments we pass to a function when we call it

```
double net_worth(double assets,  // formal parameters
                 double debt) {
  return assets - debt;
}

int main() {
 cout << net_worth(10000, 3500); // actual parameters
}
```

Positional and Named Parameters

- **positional** parameters - order of the arguments must match the order of the formal parameters
    - more concise syntax, but must pass arguments in proper order
- **named** parameters, the call can explicitly specify the name of each formal parameter (called an *argument label*) for each argument
    - more readable, less bugs with mis-ordered parameters

Default Parameters

- passing arguments with default parameters is *optional*--default value is used
- in languages w/o mandatory arg labels (like C++, python), default parameters must be at *end* of parameter list (ambiguous)
- but in languages with mandatory argument labels (like Swift), default values can be used for any parameter.

Optional Parameters

- Some languages (like Python) allow optional parameters without default values!
- A function can check if a given argument was present when the function was called and act accordingly

```
def net_worth(assets, debt,**my_optionals):  # my_optionals is a dict
    total_worth = assets - debt
    if "inheritance" in my_optionals:
        total_worth = total_worth + my_optionals["inheritance"]
    return total_worth

net_worth(10000, 2000, inheritance=50000)
```

- make code more terse, but harder to understand -- looking at the function prototype gives you incomplete information on what the parameters mean!

Variadic functions

- a function that can receive an arbitrary number of arguments
- To implement, most languages add variadic arguments to a container (array, dictionary, tuple)